



TC1030_202111

PROGRAMACIÓN ORIENTADO A OBJETOS

ORGANIZACIÓN DE LA SESIÓN

- La memoria en C++
 - Cómo se almacenan variables en la memoria
- Apuntadores
 - ¿Qué son? Variables que guardan direcciones
 - ¿Cómo se declaran? Utilizando el operador *
 - ¿Cómo se les asigna un valor? Operador “address of” &
 - ¿Cómo se obtienen los valores a los que apunta el pointer? **De-referenciación**
 - ¿Cómo se reserva memoria de manera dinámica?
- Funciones
 - Qué pasa con la memoria cuando se ejecuta una función
 - Paso de parámetros en una función
 - Pasar parámetros por **valor**
 - Pasar parámetros por **referencia**

MEMORIA EN C++

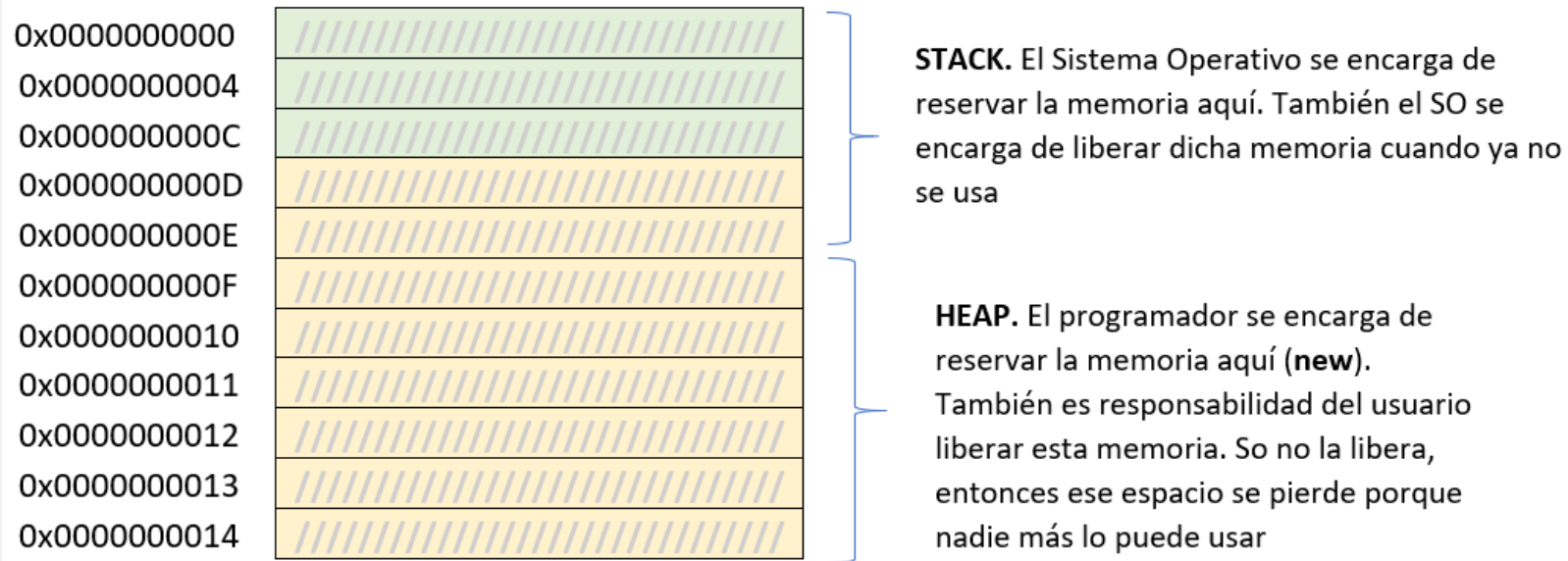
- Cómo se almacenan las variables en C++

- En direcciones de memoria fijas
- Las direcciones se representan en notación hexadecimal
- Depende del tipo de datos que se vaya a almacenar la cantidad de bytes (celdas) que se reservan (int, double, etc)
- La dirección se puede obtener utilizando el operador “address of” que se representa con ampersand **&**

0x000000000000	
0x000000000004	
0x00000000000C	
0x00000000000D	
0x00000000000E	
0x00000000000F	
0x000000000010	
0x000000000011	
0x000000000012	
0x000000000013	
0x000000000014	

APUNTADORES

- Permiten administrar la memoria Dinámica de mi programa
- La memoria está dividida en diferentes segmentos (aquí sólo aprenderemos 2)
 - **Stack** (memoria estática)
 - **Heap** (memoria dinámica)
- El Stack lo administra el S.O. (reserva y libera memoria)
- El Heap lo administra el programador ☹️



APUNTADORES

- **Definición**

- Son un tipo especial de variables
- Almacenan direcciones de memoria (**referencias** de memoria)
 - Se llaman apuntadores porque “apuntan a una dirección de memoria”

- **Declaración**

- Siguen las mismas reglas de declaración de variables
- Para declararlos se utiliza el operador * (pointer).
- Se antepone el * al nombre de la variable

```
int main()
{
    int *integerValue;
    double *doubleValue;
    char *charValue;

    return 0;
}
```

APUNTADORES

- Asignar valores de apunadores
 - Por lo pronto sólo asignaremos valores a los apunadores

```
#include<iostream>
using namespace std;

int main()
{
    int integerValue = 10;
    double doubleValue = 0.23023;
    char charValue = 'a';

    // La variable pointer2int guarda la dirección de la variable integerValue
    int *pointer2int = &integerValue;

    // La variable pointer2double apunta a la dirección la variable doubleValue
    double *pointer2double = &doubleValue;

    // La variable pointer2char apunta a la dirección de la variable charValue
    char *pointer2char = &charValue;
```

0x0000000000

0x0000000004

0x000000000C

0x000000000D

0x000000000E

0x000000000F

0x0000000010

0x0000000011

0x0000000012

0x0000000013

0x0000000014

integerValue = 10

doubleValue = 0.23023

charValue = 'a'

pointer2int = 0x0000000000

pointer2double = 0x0000000004

pointer2char = 0x000000000C

bit

bit

bit

bit

bit

APUNTADORES

- **De-referenciación.** De-referenciar significa acceder al contenido al que apunta el pointer
- Para de-referenciar se usa el mismo * que usamos para declarar un apuntador

```
void main3()
{
    int integerValue = 10;
    double doubleValue = 0.23023;
    char charValue = 'a';

    int *pointer2int = &integerValue;
    double *pointer2double = &doubleValue;
    char *pointer2char = &charValue;

    cout << " la dirección que guarda pointer2int es " << pointer2int << " y su contenido es " << *pointer2int << endl;
    cout << " la dirección que guarda pointer2double es " << pointer2double << " y su contenido es " << *pointer2double << endl;
    cout << " la dirección que guarda pointer2char es " << pointer2char << " y su contenido es " << *pointer2char << endl;
}
```

APUNTADORES

- Reservar memoria dinámica con apuntadores.
- Se utiliza el operador **NEW**

```
int main()
{
    int *pointer2int;

    pointer2int = new int;

    return 0;
}
```

0x0000000000	pointer2int = 0x000000011
0x0000000004	
0x000000000C	
0x000000000D	
0x000000000E	
0x000000000F	
0x0000000010	
0x0000000011	////
0x0000000012	////
0x0000000013	////
0x0000000014	////

```
pointer2int = new int;
```

Cuando ejecutamos la instrucción anterior se reserva espacio para almacenar un entero -4 bytes- Dicho espacio se reserva en el heap

Observa cómo el espacio se reservó al final de la memoria. Se utilizaron //// para indicar que es espacio reservado, pero nota que ese espacio NO TIENE NOMBRE.

APUNTADORES

- **Liberar memoria dinámica.** Recuerda que la memoria estática (stack) la libera automáticamente el compilador, pero la memoria dinámica(heap) NO. Es responsabilidad del programador reservarla y liberarla
- Para liberar la memoria se utiliza la palabra reservada **delete**


```
int main()
{
    int *pointer2int;
    float *pointer2float;
    string *pointer2string;

    pointer2int = new int;
    pointer2float = new float;
    pointer2string = new string;

    *pointer2int = 323;
    *pointer2float = 323.3f;
    *pointer2string = "échale ganitas para no reprobar";

    delete pointer2int;
    delete pointer2float;
    delete pointer2string;

    return 0;
}
```



Se debe liberar la memoria dinámica reservada antes de que termine nuestro programa

FUNCIONES

- Paso de parámetros en funciones. Un elemento muy importante de una función son sus parámetros ¿cómo se ven los parámetros en memoria?
- Se pueden pasar parámetros a una función de 3 formas diferentes
 - Por valor (copia) “pasar por valor”
 - Por Referencia “pasar por referencia”
 - Por Dirección “pasar por dirección”

FUNCIONES PASAR POR VALOR

- Cuando pasas un parámetro en una función se copia dicho valor en memoria (pasar por valor)

```
87  int add(int num1, int num2)
88  {
89      return num1+num2;
90  }
91
92  int multiply(int num1, int num2)
93  {
94      return num1*num2;
95  }
96
97  int main2()
98  {
99      int num1 = 10;
100     int num2 = 20;
101
102     int res = add(num1, num2);
103 }
104
```

0x000000000000
0x000000000004
0x00000000000C
0x00000000000D
0x00000000000E
0x00000000000F
0x000000000010
0x000000000011
0x000000000012
0x000000000013
0x000000000014

num1 = 10

num2 = 20

res = 30

FUNCIONES PASAR POR REFERENCIA

- Pasar parámetros por referencia significa que, en lugar de pasar y copiar un valor, pasamos la referencia/dirección de memoria. Esto evita que tengas copias de los datos por todos lados
- Se utiliza el operador **&** para pasar valores por referencia
- Nota. Ten cuidado porque cuando pasas direcciones, entonces el programador tiene permitido modificar los valores originales ¿es lo que realmente quieres?

```
int add(int &num1, int &num2)
{
    return num1+num2;
}

int multiply(int &num1, int &num2)
{
    return num1*num2;
}

int main2()
{
    int num1 = 10;
    int num2 = 20;

    int res = add(num1, num2);
}
```

Pasar por referencia es tan simple como agregarle el **&** en la lista de parámetros de la función. Todo lo demás es idéntico al código que tenemos en la sección anterior. Y claro, YA NO HAY COPIAS INECESARIAS EN MEMORIA.

¿cómo sé que el **&** es para pasar por referencia y no para obtener la dirección de una variable?

Fácil. Si el ampersand está en la firma de una función, entonces es paso por referencia, si no, entonces es para obtener la dirección de una variable

FUNCIONES PASAR VALORES POR DIRECCIÓN

- Pasar valores por dirección es lo mismo que pasar los **valores** originales.
- Para pasar los valores por dirección sólo es necesario que los parámetros sean apuntadores
- Muy importante: Los valores se pueden modificar como si fueran los originales, pero **LAS DIRECCIONES** de los valores **NO SE DEBEN MODIFICAR**

```
#include<iostream>
using namespace std;

void modify(int* value)
{
    *value = 1000;
}

int main()
{
    int value = 8;
    modify(&value);
    cout << "observa cómo en la función se modifica la variable value: " << value;
}
```

FUNCIONES PASAR VALORES POR DIRECCIÓN

PASO 1.

```
int main()
{
    int value = 8;
    modify(&value);
    cout << "observa cómo"
}
```

0x000000000	value = 8
0x000000004	
0x000000008	
0x00000000C	
0x000000010	
0x000000014	
0x000000018	
0x00000001C	
0x000000020	
0x000000022	
0x000000024	

PASO 2.

```
void modify(int* value)
{
    *value = 1000;
}
```

0x000000000	value = 8
0x000000004	value = 0x00000000
0x000000008	
0x00000000C	
0x000000010	
0x000000014	
0x000000018	
0x00000001C	
0x000000020	
0x000000022	
0x000000024	

PASO 3.

```
void modify(int* value)
{
    *value = 1000;
}
```

0x000000000	value = 1000
0x000000004	value = 0x00000000
0x000000008	
0x00000000C	
0x000000010	
0x000000014	
0x000000018	
0x00000001C	
0x000000020	
0x000000022	
0x000000024	

FUNCIONES PASAR VALORES POR DIRECCIÓN

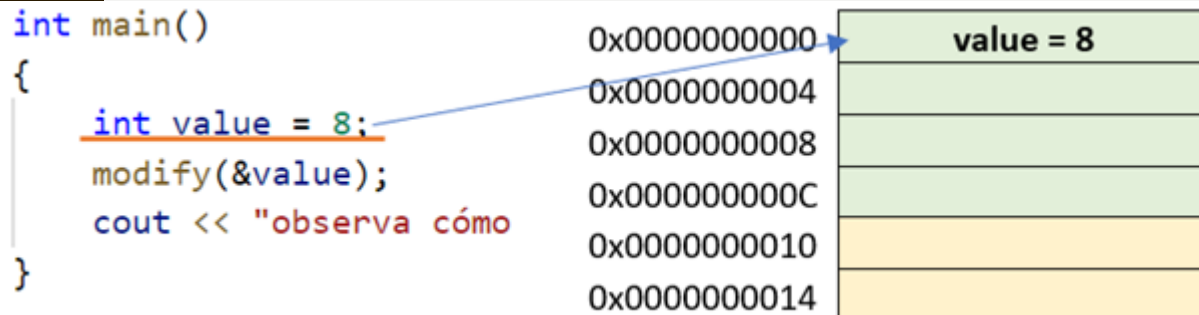
- Ahora veamos un código con lo que **no** se debe hacer (**NO SE DEBE CAMBIAR LA DIRECCIÓN DEL APUNTADOR** –más adelante veremos la forma correcta de hacerlo)

```
void modify(int* value)
{
    // ahora cambiamos la dirección a la que apunta el pointer
    value = new int;
    *value = 1000;
}

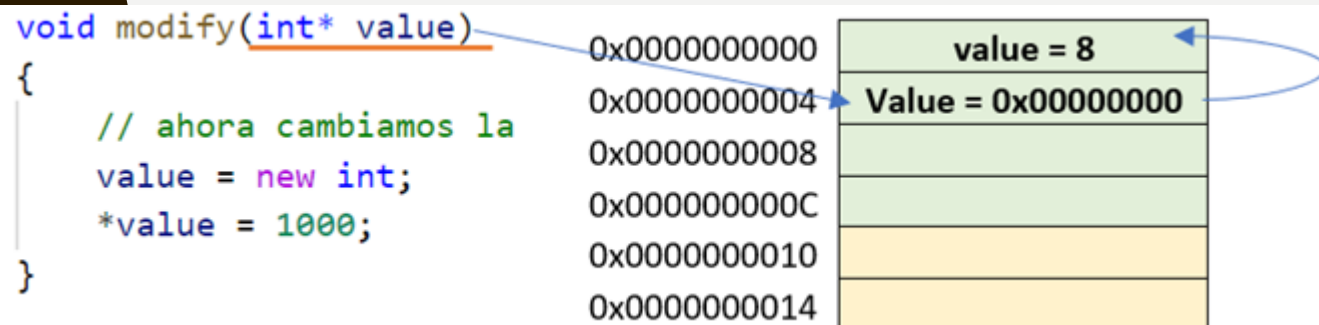
int main()
{
    int value = 8;
    modify(&value);
    cout << "observa cómo en la función se modifica la variable value: " << value;
}
```

FUNCIONES PASAR VALORES POR DIRECCIÓN

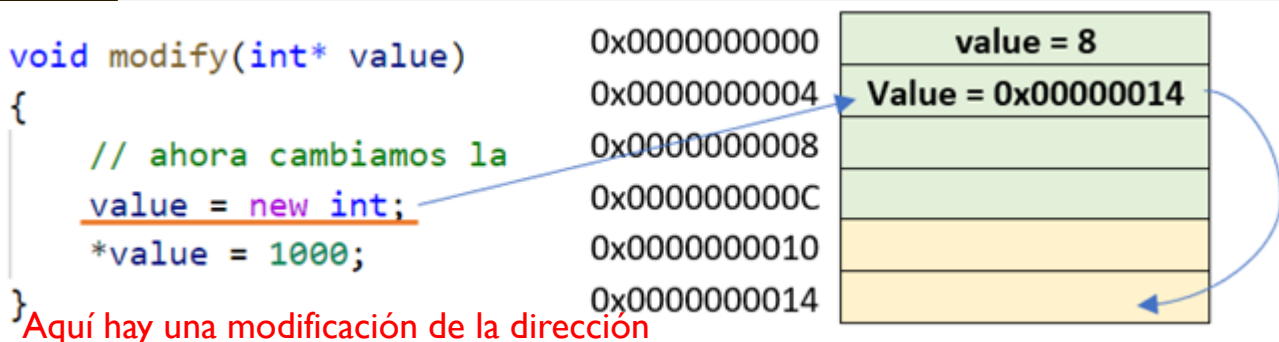
PASO 1



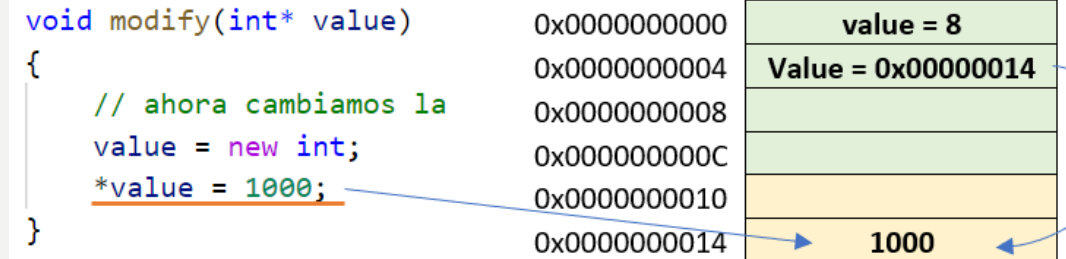
PASO 2



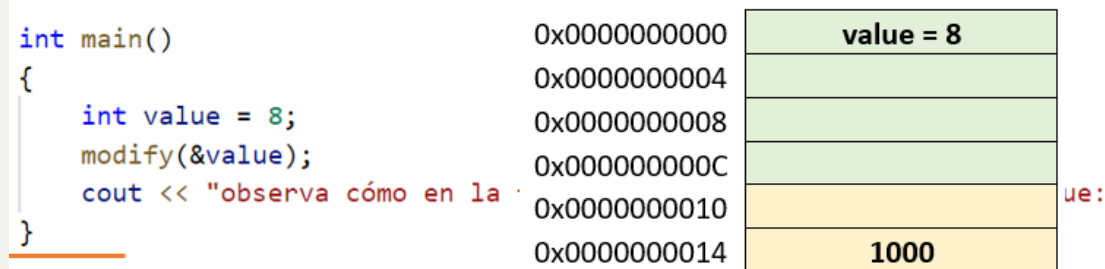
PASO 3



PASO 4



PASO 5



Aquí hay un problema (Memory Leak). Una dirección a la que nadie apunta (0x00014) y que no se libera sola pq fue creada por el usuario en el heap